**DOCKET NO.: MSFT-0681/183208.1** **PATENT**

**IN THE SPECIFICATION:**

Please replace the paragraph beginning at line 5 of page 1, with the following rewritten paragraph:

--This application is related to U.S. Patent Application No. 60/118,668, entitled "COMMON DISTRIBUTED OBJECT PLATFORM," filed on February 3, 1999; U.S. Patent Application No. 09/322,455, now patent no. 6721798, entitled "METHOD AND SYSTEM FOR TRACKING SOFTWARE COMPONENTS," filed on May 28, 1999; U.S. Patent Application No. 09/322,962, now patent no. 6834286, entitled "METHOD AND SYSTEM FOR TRACKING CLIENTS," filed on May 28, 1999; U.S. Patent Application No. 09/322,643, entitled "AUDIO VISUAL ARCHITECTURE," now patent no. 6466234, filed on May 28, 1999; U.S. Patent Application No. 09/322,459, entitled "METHOD AND SYSTEM FOR CONTROLLING ENVIRONMENTAL CONDITIONS," filed on May 28, 1999; U. S. Patent Application No. 09/322,207, now patent no. 6676934, entitled "METHOD AND SYSTEM FOR DISTRIBUTING ART," filed on May 28, 1999; U.S. Patent Application No. 09/322,964, entitled "METHOD AND SYSTEM FOR GENERATING A USER INTERFACE FOR DISTRIBUTED DEVICES," filed on May 28, 1999; and U.S. Patent Application No. 09/322,852, entitled "METHOD AND SYSTEM FOR MANAGING SOFTWARE COMPONENTS," filed on May 28, 1999, the disclosures of which are incorporated herein by reference.--

Please replace the paragraph beginning at line 13 of page 2, with the following rewritten paragraph:

--Such large distributed systems need to ensure that they can function even when portions of the system fail or go off-line for maintenance. For example, when one object goes down because of a failure of one computer, the objects that reference that failed object should not also fail. In addition, when the failed object eventually comes up, the objects that reference that failed object should be able to continue to access the object. This tracking of objects as they go down and come up can be very complex. For example, in a large distributed system, there may be no guarantee that messages relating to when an object comes up or goes down are received in the order in which they are generated or even received at all. Thus, applications accessing the objects need to perform these complex processes to ensure that references are current. Current object models, however, provide very little support for tracking objects in such complex systems.--

Please replace the paragraph beginning at line 13 of page 15, with the following rewritten paragraph:

--The watching of a resource is coordinated by the bus manager, but the monitoring of a resource is performed on a client-to-server node basis without interaction from the bus manager. When a client wants to watch a resource so that it knows when the resource is

in the up state, the client node notifies the bus manager. The resource is identified using a tracking reference. If the resource is already up, the bus manager then notifies the client node that the resource is up. Otherwise, the bus manager notifies the client node when the resource comes up. When the client node is notified that the resource is up, it may notify the bus manager to stop watching for the resource. The monitoring of a resource is performed on a peer-to-peer basis. That is, once a client node is informed that a resource has entered the up state, it establishes a connection directly with the server node that contains the resource. Once the connection is established, the client node notifies the server node periodically that it is still up and running. If the server node does not receive this notification, it assumes that the client node is no longer up and running and resets its internal state accordingly. Similarly, if the client node receives an error when sending its periodic notification to the server node, it assumes the server node is down and resets its internal state accordingly. When the resource goes down, the server node notifies the client node that the resource is now down. Each node includes clients 205, a resource tracking system 206, and a resource manager 207. A client requests the resource tracking system to provide pointers to resources and to notify the client when resources come up or go down. The resource tracking system interacts with the resource manager to watch, monitor, and retrieve pointers to the resource. The resource manager also detects when resources on its node come up and go down and notifies the bus manager or other nodes as appropriate. The nodes may all be computer systems with a central processing unit, memory, and input/output devices. The software components and data structures of these nodes may be stored on computer-readable medium such as memory, CD-ROM, flexible disk, hard disk, and so on and may be transmitted via a data transmission medium.--

Please replace the paragraph beginning at line 1 of page 17, with the following rewritten paragraph:

--Figure 2B is a block diagram illustrating the watching of a resource. The client node 2B10 is a node on which a client 2B11 has requested to watch for a certain resource to come up. The client node tracks the resource with components 2B12 of the resource manager. The resource manager includes a watched resource table 2B13, a process/resource list 2B14, and resource directory 2B15. The client and the resource manager interact as described in Figure 2A. The resource manager interacts with a bus manager 2B20 to notify the bus manager when resources of that node go up and come down and to request that the bus manager watch for a resource on the client node's behalf. The bus manager includes a bus manager component 2B21 that has a watched resource table 2B22 and a resource directory 2B23. The watched resource table at the client node contains an indication of each resource that a client located at that node has requested to watch along with an indication of the requesting client. The resource directory at the client node contains the identification of each resource that is currently up at the client node. The resource manager uses the resource directory to re-notify the bus manager of the resources that are up in the event that the bus manager goes down and then comes back up or when another bus manager takes control of the bus. The resource manager similarly uses the watched resource table to re-notify the bus manager of the resources that its clients are watching.

The process/resource list identifies each resource by the process in which it is executing. When a process goes down, the resource manager can use the process/resource list to notify the bus manager that those resources are now down. The client node sends to the bus manager an attach resource message whenever a resource comes up and a detach resource message whenever a resource goes down. The client node sends a watch resource message to the bus manager to notify the bus manager to start watching for a particular resource to come up. The client node keeps track of the resources that it is watching in the watched resource table. If a client requests to watch a resource that another client on the same node is already watching, then the client node does not need to send another watch resource message to the bus manager. The client node sends a stop watching resource message to the bus manager whenever it wants to stop watching for a resource to come up. The client node may want to stop watching for a resource whenever all of its clients who were watching the resource request to stop watching the resource or whenever the resource comes up. The client node sends a find resource message to the bus manager when it wants to retrieve a pointer to a resource. When a client comes up, the bus manager sends a watched resource is up message to each client node that is watching that resource. The watched resource table of the bus manager contains an identifier of each resource that is being watched and the client node that requested a watch of that resource. The resource directory of the bus manager contains an identifier of and a pointer to each resource that is currently up. When the bus manager receives an attach resource message, it updates the resource directory to indicate that the resource is up. It then checks the watched resource table to determine whether any client nodes are watching that resource. If so, the bus manager sends a watched resource is up message to each such client node. When the bus manager receives a detach resource message, it updates its resource directory to indicate that the resource is now down. When the bus manager receives a node is down message, it effectively detaches all resources that were attached at the node that is now down and effectively stops all the watches from that node.--

A4 Concl.

Please replace the paragraph beginning at line 2 of page 20, with the following rewritten paragraph:

--Figure 3 is a block diagram illustrating data structures of the resource tracking system. The resource tracking system provides a directory object 301, a list of resource reference objects 302, and, for each resource reference object, a list of client objects 303. The directory object provides functions for controlling the access to resources by interacting with an installable resource manager. The directory object maintains a resource reference object for each resource that a client has registered to track. Each resource reference object contains the name of the resource and, if the resource is up, a unique instance identifier of the resource (*e.g.*, a pointer to the resource and the time at which the resource was created). Each resource reference object also contains a pointer to a client list of client objects that each represent a client that has registered to track the corresponding resource. Whenever a client wants to reference to a resource, it supplies a client object to the resource tracking system. This client object includes functions that the resource tracking system uses to notify the client when the resource changes its state (*e.g.*, a call-back routine). Since these data structures may be accessed concurrently by multiple threads

A5

A5 cont

of execution, a concurrency management technique is used when accessing these data structures. For example, before accessing a data structure, a thread may lock the data structure and then unlock it after the access is complete. In the following, the description of the functions that access these data structures omit these well-known concurrency management techniques.--

Please replace Table 1 beginning at line 10 of page 21, with the following rewritten Table 1:

**Table 1**
**Directory Object**

| Name | Description |
|---|---|
| myResRefList | A pointer to the resource reference list for this directory object. |
| addNewResClient | A function that is invoked by a client to register that it wants to track a resource. |
| resIsUp | A function that is invoked by the resource manager to notify the resource tracking system that a resource is up. |
| resIsDown | A function that is invoked by the resource manager to notify the resource tracking system that a resource is down. |
| busStateChanged | A function that is invoked by the resource manager to notify the resource tracking system that the bus has changed state (*i.e.*, came up or gone down). |
| reevaluateRefs | A function that is invoked by the resource manager to notify the resource tracking system to reset all its references to the down state. |
| clearInitialBlock | A function that is invoked by the resource manager to notify the resource tracking system to start processing notifications. |
| goingAway | A function that is invoked by a resource reference object to notify the directory object that a resource reference object is going away. |
| monitorRes | A function that is implemented by the resource manager and invoked by the resource tracking system to notify the resource manager to start monitoring a resource to go down. This function may be provided as a derivation of the directory object. |
| stopMonitoringRes | A function that is implemented by the resource manager and invoked by the resource tracking system to notify the resource manager to stop monitoring a resource to go down. This function may be provided as a derivation of the directory object. |
| watchRes | A function that is implemented by the resource manager and invoked by the resource tracking system to notify the resource manager to start watching for a resource to come up. This function may be provided as a derivation of the directory object. |
| stopWatchingRes | A function that is implemented by the resource manager and invoked by the resource tracking system to notify the resource manager to stop watching for a resource to come up. This function may be provided as a derivation of the directory object. |
| finders | A function that is implemented by the resource manager and invoked by the resource tracking system to retrieve a pointer to a resource. This function may be provided as a derivation of the directory object. |

A6

Please replace the paragraph beginning at line 2 of page 23, with the following rewritten paragraph:

A7

--A pointer used may be "smart pointer" such that when a pointer is copied it is automatically reference counted; when the pointer is reset, it is automatically released. When a smart pointer goes out of scope, its destructor releases it. The myResPtr of the resource reference object and myRefResPtr of the client object are smart pointers.--

Please replace the paragraph beginning at line 5 of page 26, with the following rewritten paragraph:

A8

--Figures 14-19 are flow diagrams illustrating the functions of the directory objects. Figure 14 is a flow diagram of an example implementation of the Directory::addNewResClient function. This function adds a new client object for a resource to the directory. This function is passed the name of the resource (resName) and the client object. In step 1401, the function finds the resource reference object associated with the passed resource name by invoking the findRefRes function of the directory object. That function searches the resource reference list and returns a reference to a resource reference object with that name. In step 1402, if a resource reference object with that name is found, then the function continues that step 1407, else the function continues at step 1403. In steps 1403-1406, the function adds a resource reference object for the named resource to the resource reference list. In step 1403, the function creates a resource reference object. In step 1404, the function adds the resource reference object to the resource reference list of this directory object. In step 1405, the function adds the client object to the client list of the resource reference object by invoking the add function of the resource reference object passing the client object. In step 1406, the function signals the clients that the resource is up by invoking the up function of the resource reference object. If the resource is not actually up, the resource tracking system will enter the watch for resource state for this resource and notify clients that the resource is down. In step 1407, the function adds the client object to the client list of the resource reference object by invoking the add function of the resource reference object. In step 1408, the function initializes the client object by invoking the init function of the resource reference object passing the client object and then returns.--

Please replace the paragraph beginning at line 11 of page 35, with the following rewritten paragraph:

A9

--Figure 31-34 are flow diagrams illustrating the processing of the functions of the client object. Figure 31 is a flow diagram of an example implementation of the Client::resIsUp function. This function is invoked when the resource comes up. In step 3101, the function sets the interface pointer of this client object to null. In step 3102, the function retrieves a reference

A9 cont.

counted pointer to the resource by invoking the getRefCountedPtr function of the resource reference object and saves it in a smart pointer. In step 3103, if the client object has an interface identifier specified, then the function continues at step 3104, else the function continues at step 3105. In step 3104, the requested interface pointer is retrieved from the subject resource. In step 3105, the function retrieves a pointer to the interface by invoking the query interface function of the resource. In step 3105, the function notifies the client derivation that the resource is now up by invoking the resourceIsUp function provided by the client. The function then returns.--

Please replace the paragraph beginning at line 1 of page 36, with the following rewritten paragraph:

--Figure 33 is a flow diagram of an example implementation of the Client::deleteYourself function. This function is invoked when this client object is to be deleted. In step 3301, the function clears the interface identifier of this client object. In step 3302, if this client object has a reference to a resource reference object, then the function continues at step 3303, else the function continues at step 3308. In step 3303, the function notifies the resource reference object that this client is going away by invoking the to be goingAway function. In step 3304, if the invocation is successful, then the function continues at step 3305, else the function continues at step 3306. In step 3305, the function decrements the reference count for the resource reference object and sets its pointer to null. In step 3306, the function sets the delete flag of this client object to true. In the step 3307, if this client object has a reference to a resource reference object, then the function returns, else the function continues at step 3308. In step 3308, the function destructs this client object and returns.--

A10

Please replace the paragraph beginning at line 14 of page 36, with the following rewritten paragraph:

--Figure 34 is a flow diagram of example implementation of the Client::resourceIsUp function. This function is provided by a client to specify client specific processing to be performed when a resource comes up. In step 3401, this function sets a resource is up a flag for the client. The client also provides an analogous function for when a resource goes down.--

A11

Please replace the paragraph beginning at line 23 of page 36, with the following rewritten paragraph:

-- Figures 35-39 are flow diagrams of functions of a resource manager for watching a resource. Figure 35 is a flow diagram of an example implementation of the ResourceUp function of the resource manager. The resource manager invokes this function whenever a resource at that node is detected as being up. The resource manager may know that a resource is up because it controlled the creation of the resource at startup, because it dynamically created the resource when requested by another resource, or because another resource created that

A12

A12

resource and registered the created resource with the resource manager. This function is passed a pointer to the resource that is now up. In step 3501, if the bus is up, then the function notifies the bus manager by invoking the attach function of the bus manager passing the identification of the resource that is now up, else the function returns. In step 3502, the function updates the local resource directory to indicate that the passed resource is up. The function then returns.--

Please replace the paragraph beginning at line 11 of page 41, with the following rewritten paragraph:

A13

--Figure 45 is a flow diagram of an example implementation of the Directory::monitorRes function. This function is implemented by the resource manager and is invoked by a client to start monitoring for the passed resource to go down. In step 4501, the function uses the passed resource to identify the server node (*i.e.,* the node where the resource is located) for the resource. The function may use the query interface function of the resource to retrieve a pointer to an interface for identifying the server node. The function also updates the monitor resource table for the resource and client so that the client can be notified when the resource goes down and so that the server node can be re-notified if it goes down and comes up. In step 4502, if a connection has already been established with the server node as indicated by the client connection table, then the function continues at step 4508, else the function continues at step 4503. In step 4503, the function establishes a connection with the server node by invoking the connectClient function of the server node passing a pointer to an interface of the client node for receiving notifications and passing a client context for identifying this connection. The invocation returns a server context. In step 4504, if an error is detected when invoking the connectClient function, then the function continues at step 4505, else the function continues at step 4506. In step 4505, the function assumes that the server node is down and performs the associated processing and then returns. In step 4506, the function updates the client connection table to indicate that a connection has now been established with server node with the client and server context. In step 4507, the function signals to start sending a client is alive message periodically to the server node. The sending of the client is alive message may be considered to be one form of "leasing" a resource. In step 4508, the function invokes the monitor resource function of the server node passing the identification of the resource to be monitored. The function then returns.--

Please replace the paragraph beginning at line 5 of page 46, with the following rewritten paragraph:

A14

--The tracking system also provides for watching the properties of a server resource by a client resource. A property of a resource corresponds to data related to the resource whose value can be set by that resource during execution of a function of the resource. That function can be invoked by another software component. The function may be specifically provided to set the value of the property (*e.g.*, a set property function) or may set the value of the property as a side effect. In one embodiment, a property watching component of the resource

tracking system allows client resources to register their interest in receiving notifications when a property of a server resource is set. The client resources also specify the behavior to be performed when the property is set. The property watching component provides a synchronous mechanism for notifying client resources when the property is set. This synchronous mechanism ensures that client resources who are registered to watch a property are notified of the setting of the property before any client resources are notified of a subsequent setting of the property. The property watching component thus provides a mechanism for synchronizing processing among multiple client resources.--

Please delete the paragraph beginning on line 19 on page 48.

Please replace the paragraph beginning at line 1 of page 51, with the following rewritten paragraph:

--Figure 61 is a flow diagram of an example implementation of a register watch function of a client resource. This function is passed the identification of the server resource, the name of the property to be watched, and identification of the client resource. In step 6101, if the client resource is already watching that property, then the function continues at step 6106, else the function continues at step 6102. In step 6102, the function creates a unique context for the client resource and property. In step 6103, the function invokes the watch property function of the server resource passing the name of the property and the context. In step 6104, the function creates a property client object for that property and adds that object to the client object list of the resource reference object for that resource. In step 6105, the function adds an entry to the context/property table. In step 6106, the function adds a property reference object to the list associated with the property client object and then returns.--

Please replace the paragraph beginning at line 24 of page 51, with the following rewritten paragraph:

--The event system provides a mechanism for providing event notifications when events are generated by resources. An event is an asynchronous signal that is distributed to all client resources, also referred to as listeners, who have registered to listen for an event signal. In one embodiment, the event system neither guarantees that a listener will receive the events in the order they are generated nor guarantees that each listener will receive every event for which it is listening. Each event has an associated event type. A listener registers to listen for events of a certain event type. In one embodiment, the event types may be hierarchically organized. For example, one event type may be a timer event. The timer events may be further classified into catastrophic timer events, warning tuner events, and informational timer events, which are sub-events. An informational timer event may further be classified into start-up timer events and shut-down timer events. A listener may register to listen for events at any level in the event hierarchy. For example, a listener may register to listen for informational timer events. That listener would receive an event notification as for start-up tuner events and a shut-down timer

A16 concl.

events. A listener will receive event notifications for leaf events of the sub-tree corresponding to the event type registered. A leaf event is an event that is not further classified into sub-events. An event type may have its hierarchy embedded in its name. For example, the name of start-up timer event may be "/timer event/informational time event/start-up timer event."--

Please replace the paragraph beginning at line 16 of page 52, with the following rewritten paragraph:

--Figure 63 is a block diagram illustrating components of the event system in one embodiment. A client 6301 registers to listen for events by sending a listen message along with an event type to the listener component 6303. The client receives from the listener component an event notify message along with event information when an event of that event type is generated. The client un-registers its interest in listening for events of a certain event type by sending a stop listening message along with the event type to the listener component. In one embodiment, each node has a listener component through which is routed all event-related messages for all listeners on that node. The listener component may in turn route event-related messages to a listener bus manager 6305. The listener component notifies the listener bus manager to listen for all event types for which listeners on that node have registered. The listener component may send only the listener bus manager. There is one listen message for each event type regardless of how many listeners at that node have registered for that event type. For example, if a listener component receives requests from six clients, the listener component sends only one listen message to the listener bus manager. The listener component maintains a listener table cache 6306 that contains a mapping from each event type for which a listen request has been registered and each client that has registered for that event type. When the listener component receives an event notification, it uses the listener table cache to notify each listener that has registered for that event type. In this way, the listener component reduces the event messages that are sent between that node and the node of the listener bus manager. When the listener component receives event notifications, it queues an event notifications for each of the listeners. The listener component uses a separate thread for providing the event notification to each listener. If a single thread were used to notify each listener, the event notifications could be delayed to some listeners as a result of a delay or problem in notifying another listener. The use of a separate thread for each listener ensures that the notification to one listener will not be delayed as a result of an event notification to another listener. The listener component may receive a bus state change message. If the bus goes down and then comes back up, the listener component can reregister with the listener bus manager to receive the events of the event types in its listener table cache. The listener component may also optimize its sending of listen requests based on the event hierarchy. For example, if a listener registers to listen for a informational timer, the listener component will register that request with the listener bus manager. If another listener registers to listen for a start-up timer, then the listener component will not need to register that request with the listener bus manager. Since the listener component has already registered to receive a higher-level event type, it is already registered to receive all lower-level event types.--

A17

Please replace the paragraph beginning at line 25 of page 54, with the following rewritten paragraph:

A18

--There are various components in the system that are responsible for the collecting, storage, and possible forwarding of log records. All forms of this component are modeled by the HcsLogFacility class.--

Please replace the paragraph beginning at line 2 of page 55, with the following rewritten paragraph:

A19

--The HcsLogFacility class is an HcsResource derivation whose purpose is to provide storage and forwarding at various levels in the system. There are currently two implementations: the Local Log Facility, and the Central Log Facility. The HcsLogFacility interface is:--

Please replace the paragraph beginning at line 20 of page 55, with the following rewritten paragraph:

A20

--A Local Log Facility (LLF) instance (only one!) exists in every node in the system. The purpose of this HcsLogFacility implementation is to accept all log records from various HCS components on that node and to buffer them in a large circular on disk until they can be delivered to the Central Log Facility (CLF). The intent is that a node could survive for some time if the CLF were to be unavailable. One of the configuration parameters passed to each Local HcsLogFacility instance will be the resource name of the HcsLogFacility through which that instance is to forward its records. The current thinking is that this will be the name of the CLF, but this is not an architectural requirement. In other words, there could be intermediate levels of HcsLogFacilities placed in the system. In fact, an LLF could be configured to forward its records to another LLF. After all, an HcsLogFacility is a HcsLogFacility.--

Please replace the paragraph beginning at line 9 of page 56, with the following rewritten paragraph:

A21

--So the question is: how do HCS components log their information? To provide a standard and safe access to the HcsLogFacilities, there is a class family provided. The family is based at a class called HcsLogPort. This class implements the common behavior of all types of LogPorts and is functional on its own.--

Please replace the paragraph beginning at line 13 of page 56, with the following rewritten paragraph:

A22

--Derived from the HcsLogPort are the HcsResourceLogPort and HcsLogFacilityPort classes. The HcsResourceLogPort provides easy support for the resource